Preprint of article to appear in Transactions of Aspect-Oriented Software Development, Vol. 1, No. 1.

Directives for Composing Aspect-Oriented Design Class Models

Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, G. Georg
Contact Email: ghosh@cs.colostate.edu

Computer Science Department
Colorado State University
Fort Collins, CO 80523

Abstract. An aspect-oriented design model consists of a set of aspect models and a primary model. Each aspect model describes a feature that crosscuts elements in the primary model. Aspect and primary models are composed to obtain an integrated design view. In this paper we describe a composition approach that utilizes a composition algorithm and composition directives. Composition directives are used when the default composition algorithm is known or expected to yield incorrect models. Our prototype tool supports default class diagram composition.
Keywords: Aspect Oriented Modeling, Composition directives, KerMeta, Metamodel, EMOF, Signature, UML.

1 Introduction

Design features that address dependability concerns (e.g., security and fault tolerance concerns) may crosscut many elements of a design model. The crosscutting nature of these features can make understanding, analyzing, and changing them difficult. This complexity can be better managed through the use of aspect-oriented modeling (AOM) techniques that support separation and composition of crosscutting features [1].

In the AOM approach that we developed [1], an aspect-oriented design model consists of a primary model and one or more aspect models that each describes a feature that crosscuts the primary model. Aspect models are generic descriptions of crosscutting features that must be instantiated before they can be composed with the primary model. An integrated view of an aspect-oriented design model is obtained by composing the instantiated aspect models and the primary model. Instantiated aspect models and primary models consist of UML [2] models. Composition of the models involves merging UML models of the same types. For example, the class model in an instantiated aspect model is merged with the class model in a primary model. In previous work, a name-based composition procedure was used to merge UML models [1]. Model elements with the same name are merged to form a single element in the composed model. The composition procedure assumes that elements with the same name represent consistent views of the same concept. This may not always be the case. For example, consider an aspect-oriented design consisting of a primary model that describes a class representing a server that provides unrestricted access to services via operations in the class, and an instantiated aspect model that describes the same server class with access control features. In this case, simple name-based merging of the two classes and the operations in them could lead to operations that are associated with inconsistent specifications (a primary model operation and its corresponding aspect model operation would have the same name but different argument lists and specifications). Often, a more sophisticated form of composition is needed to produce composed models with required properties. To meet this need we proposed the use of composition directives to ensure that the name-based composition procedure produces desired results [3].

This paper extends previous work by introducing (1) a more general form of model element matching that is based on the notion of model element signatures, (2) a composition metamodel with behavioral features that specify how UML elements are composed, and (3) new forms of composition directives. In this paper we illustrate how a signature-based composition procedure can be used to compose class models and describe how composition directives can be used to ensure that the composition procedure produces desired results.

We have developed a prototype tool that implements the class model composition behavior specified in the composition metamodel [4].

The remainder of the paper is organized as follows. Section 2 gives an overview of signature-based model composition and composition directives. Section 3 describes the composition metamodel. Section 4 describes the composition directives and provides illustrations of their use. Related work is discussed in Section 5 and Section 6 presents conclusions and plans for future work.

2 An Overview of Signature-Based Model Composition

A primary model in an aspect-oriented design model consists of one or more UML models that each describes a view of the core functionality. The core functionality determines the dominant structure of a design. Aspect models consist of UML model templates that describe generic forms of crosscutting features as patterns. An aspect model must be instantiated to produce a model that can be composed with a primary model. An instantiation of an aspect model, called a *context-specific aspect model*, describes the form the feature takes in a part of the design. Instantiating an aspect model involves binding the aspect model's template parameters to application-specific values.

A single aspect model may have to be instantiated multiple times for a given application. For example, consider the case where a decision has been made to make an application design fault tolerant and highly available by replicating critical resources such as data repositories and service providers. Incorporating the crosscutting replication feature into the (primary) design model proceeds as follows:

1. An aspect model describing the replication feature for a generic resource is developed or acquired.
2. The replication aspect model is instantiated multiple times. Each instantiation is a context-specific aspect model that describes the replication feature for a specific application resource.
3. The context-specific aspect models are composed with the primary application model to produce a design in which specified resources are replicated.

In our previous work we developed a composition procedure that used model element names to identify the elements that are to be merged. Model elements of the the same syntactic type and with the same name are merged to form a single model element. Naming conflicts can be avoided if there is a managed namespace from which values used to bind aspect models and to name primary model elements are obtained. We refer to such a namespace as the *application domain namespace* [1]. Unfortunately a managed namespace is often not available in design development environments, and thus naming conflicts may occur.

2.1 Matching Model Elements using Signatures

Name-based composition is relatively easy to implement but as a matching criterion, it can be too permissive in some cases. For example, matching operations using only their names could lead to merging problems when the operations have incompatible return types or when the argument lists differ. Similarly, matching attributes using only their names can lead to merging problems when the types associated with the attributes are incompatible. One would like to have matching criteria that take into consideration additional properties of the elements being matched. For example, one should be able to express a matching criterion for attributes that requires matching attributes to have the same name and type. The need for finer-grained matching criteria led to the development of the signature-based composition approach described in this paper.

The signature-based composition procedure merges information in model elements with matching signatures to form a single model element in the composed model. A model element's signature is defined in terms of its syntactic properties, where a syntactic property of a model element is either an attribute or

an association end defined in the element's UML metamodel class. For example, *isAbstract* is a syntactic property defined in the metamodel class called `Class`. If an instance of `Class` is an abstract class then $isAbstract = true$ for the class, otherwise the instance is a concrete class (i.e., $isAbstract = false$).

The signature of a model element is a collection of values for a subset of syntactic properties defined in the model element's metamodel class. The set of syntactic properties used to determine a model element's signature is called a *signature type*. For example, the signature type for an operation can be defined as a set consisting of the following properties defined in the `Operation` class: `name` (value is the operation's name) and `ownedParameter` (value is the collection of parameters associated with the operation). Using this signature type, the signature of an operation $update(x : int, y : int)$ is the set $\{update, (x : int, y : int)\}$. If this signature is used to match operations, two operations match if and only if they have the same name and parameter list. If the signature type of an operation consists only of the operation name, then the signature of the operation is $\{update\}$. Use of this name-only signature type results in a weaker matching criterion for operations: two operations match if and only if they have the same name.

A signature type that consists of all syntactic properties associated with a model element is called a *complete* signature type. Complete signature types require that matching model elements have equivalent values for all syntactic properties (i.e., the matching elements must be syntactically identical). Complete signature types are typically used for matching contained model elements such as class attributes and operation parameters. Composite model elements that contain a variety of model elements (e.g., classes) tend to have signature types that are not complete.

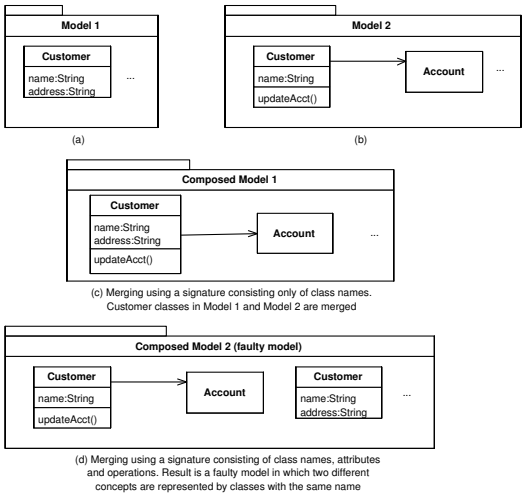

Fig. 1. An Example of Model Element Matching and Merging

If two model elements of the same syntactic type[1] have the same signature, then their properties are merged to form a single model element of that syntactic type. As an example, consider a model, *Model 1*, containing a concrete class named *Customer* with attributes *name* and, *address*, (see Fig. 1(a)) and another model, *Model 2*, which contains a concrete class named *Customer* with an attribute *name* and a reference to an *Account* object (see Fig. 1(b)). If the signature type used to compose the classes in Fig. 1(a) and Fig. 1(b) consists of the class name property and the *isAbstract* property then the two classes match (they have the same

[1] The syntactic type of a model element is the class of the model element in the UML metamodel

name and they are both concrete) and their contents are merged to form a single class. The issue of merging syntactic properties that are not part of a model element's signature type arises in this case. The matching classes in this example have different attribute, operation and association end sets. Merging the constituent model elements involves matching them using signature types defined for the elements. The constituent elements that are matched are merged in the composed model. Those elements that are not matched are included in the composed model.

The composed model shown in Fig. 1(c) is obtained by using complete signature types for attributes, operations and association ends:

- The attribute *name* : *String* in *Model 1* and *Model 2* match and is included once in the composed model.
- The attribute *address* : *String* in *Model 1* does not appear in *Model 2* and thus is not matched. It appears in the composed model.
- The operation *updateAcct*() in *Model 2* does not appear in *Model 1* and thus is not matched. It appears in the composed model.
- The association and the class *Account* in *Model 2* do not appear in *Model 1* and thus are not matched. They are included in the composed model.

The use of particular signature types can lead to models that are not syntactically well-formed in some cases. For example, consider the case in which the signature type for class is defined as consisting of the following properties: `Name`, `isAbstract`, and `ownedAttribute`. Two classes match using this signature type if and only if they have the same name, are both abstract or are both concrete, and they have the same set of attributes and association ends. If this signature type is used to compose the class models shown in Fig. 1(a) and Fig. 1(b), then the result is shown in Fig. 1(d). The model is not well-formed because there are two classes with the same name in the same namespace.

To resolve the above problem one must understand the intent behind the signature type. If it is determined by the modeler that the signature type correctly reflects the syntactic form of classes that represent the same concept, then the problem is resolved by renaming either the *Customer* class in *Model 1* or the *Customer* class in *Model 2*. As will be described later in this paper, this can be accomplished by using a `rename` composition directive. On the other hand, if the modeler determines that the classes actually represent similar classes then the signature type must be changed so that the classes are matched.

2.2 Identifying and Using Composition Directives

The composition approach that we have developed utilizes a signature-based composition algorithm and composition directives. In some cases, sole use of the algorithm will produce models with undesirable properties. This is the case when the views described by the models contain inconsistent information. In some cases, the problems can be resolved by syntactically tweaking the models that are involved in the composition or by overriding some of the composition rules. Composition directives can be used for these purposes.

Fig. 2 shows activities related to identifying and using composition directives. The activity diagram shows how the relationship among three activities: the composition activity (*Compose aspect and Primary models*), the model analysis activity (*Analyze Composed model*) and the directives identification activity (*Identify Composition Directives*). The composition activity, *Compose aspect and Primary models*, takes in three inputs: a primary model, a non-empty set of context-specific aspect models, and a (possibly empty) set of composition directives. In this activity, the aspect and primary models are composed using the algorithm and composition directives to produce a *Composed model*. The matching and merging procedure used by the composition algorithm is capable of detecting conflicting syntactic property values associated with matching model elements. For example, if two matching classes have different values for the *isAbstract* property, a conflict is flagged.

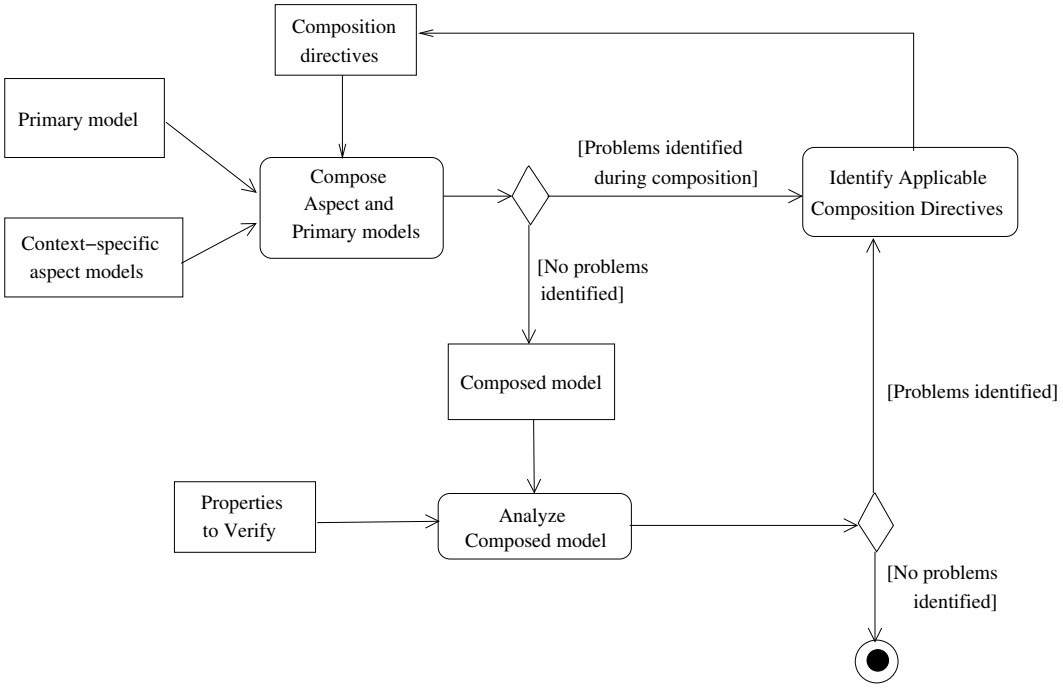

Fig. 2. Using composition directives to resolve composition problems

After composition, the composed model can be formally analyzed against desired properties (referred to as *Properties to Verify* in Fig. 2) to uncover design errors. For example, one can analyze the models against well-formedness rules to identify badly formed models or one can analyze the models against desired semantic properties (e.g., "only the owner of a file can delete the file"). In related work, we developed a technique for uncovering semantic problems during composition [5]. In the approach, the semantic property to be verified is used in the composition process to generate proof obligations. Establishing that a composed model has the stated semantic properties requires discharging the proof obligations.

In some cases, the uncovered problems can be resolved using composition directives. In these cases an appropriate set of directives are identified and used to compose the context-specific aspect and primary models. In other cases, more substantial changes may be required. For example, it may be determined that another variant of the aspect model is needed or that the primary model has to be significantly refactored.

This paper focuses on the *Compose Aspect and Primary models* activity shown in Fig. 2. Activities related to analysis of models to uncover problems and the identification of composition directives is not within the scope of this paper.

2.3 Examples of Applying Composition Directives

Composition directives can be classified as *Model Directives* and *Element Directives*. Model directives are used to determine the order in which multiple aspect models are composed with a primary model. Element directives are used to determine how an aspect model is composed with a primary model. Element directives can be classified in terms of when they are applied in the composition process:

– *Pre-Merge Directives*: These directives are used to carry out simple modifications of the models before they are merged. For example, one can rename model elements, delete model elements, or replace model elements (delete and add model elements) in the primary or context-specific aspect models.
– *Merge Directives*: These directives are used to override rules for merging model elements. For example, one can specify that a model element in one model completely replaces an element in another model.
– *Post-Merge Directives*: These directives are used to carry out simple modifications on the model produced after merging possibly modified primary and context-specific aspect models. The directives for renaming, adding, deleting, and replacing model elements also fall into this category.

In the remainder of this section we provide examples of composition problems that can be resolved using composition directives. It is important to note that the composition approach discussed in the following sections does not provide systematic techniques for analyzing composed models nor for identifying appropriate composition directives once problems are uncovered. As stated earlier, the composition algorithm will flag cases where conflicting syntactic properties exist for model elements that are merged. It does not, however, detect semantic conflicts that can arise as a result of inconsistent specifications of behavior or other semantic properties. Uncovering such semantic properties requires formal semantic analysis of the composed model.

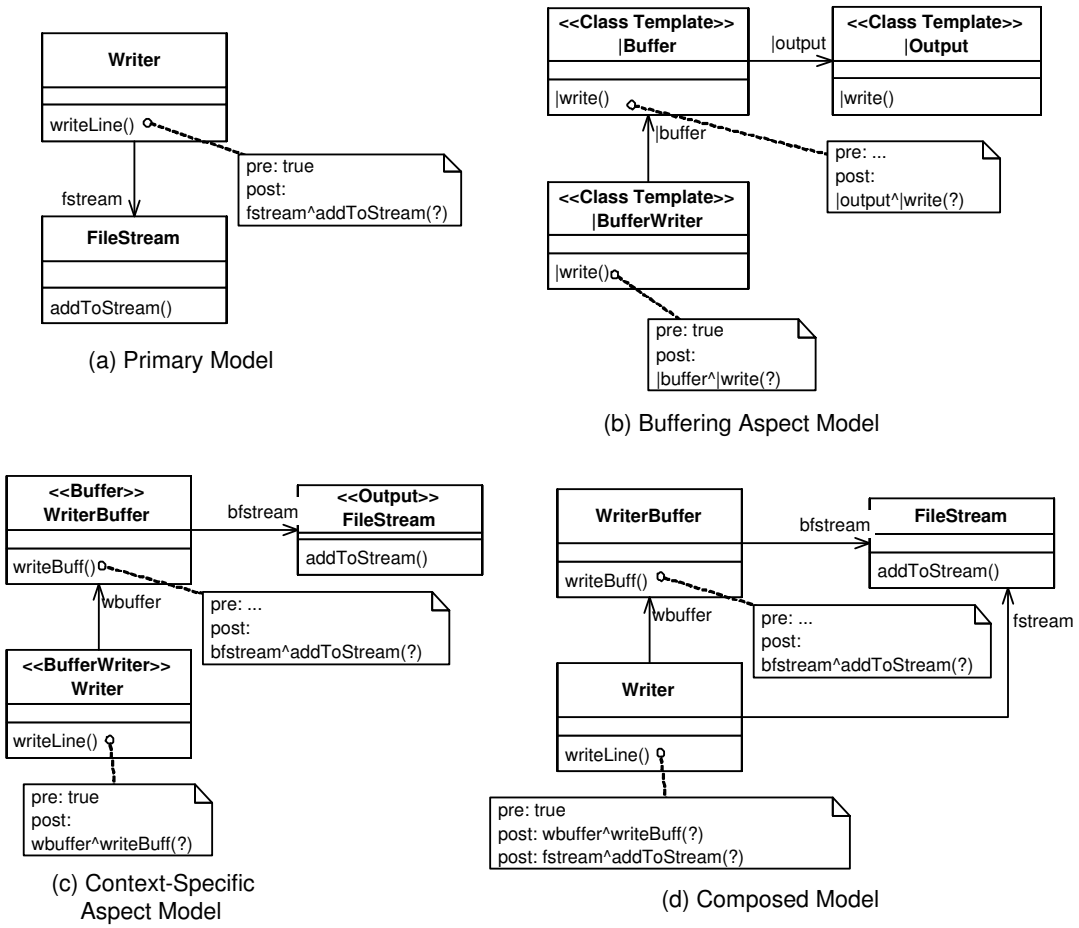

Fig. 3. An Example of a Faulty Composition

Fig. 3 shows a simple example of a composition that leads to a faulty composed class model. In the example, a modeler creates a primary model (see Fig. 3(a)) in which an output producer (an instance of `Writer`) sends outputs directly to the output device it is linked to (instance of `FileStream`). The modeler then decides to incorporate a buffering feature into the model by instantiating a buffering aspect model. Fig. 3(b) shows the class diagram template that is part of the buffering aspect model. The aspect model describes how entities that produce outputs (represented by instantiations of *BufferWriter*) are decoupled from output devices through the use of buffers . Template parameters are preceded by the symbol "|". The operation templates `|write()` in `|Buffer` and `|BufferWriter` are associated with template forms of operation specifications [1].

To incorporate the buffering feature into the primary model, the modeler must first instantiate the aspect model to produce a context-specific model. Instantiating the buffering class diagram template produces a class diagram that describes how buffering is to be accomplished in the context of the primary model. The class diagram shown in Fig. 3(c) is obtained from the buffering class diagram template using bindings that include the following:

```
(|Buffer<-WriterBuffer), (|Output<-FileStream), (|BufferWriter<-Writer),
(|BufferWriter::|write()<-writeLine()), (|Buffer::|write()<-writeBuff()),
(|Output::|write()<-addToStream())
```

The result of composing the class diagram shown in Fig. 3(c) with the primary model class diagram shown in Fig. 3(a) is presented in Fig. 3(d). Composition is carried out by matching model elements using signatures consisting only of model element names. If the matching model elements are associated with invariants, the invariant associated with the merged element in the composed model is the conjunction of the invariants in the matched elements. Operation specifications, expressed as OCL pre and postconditions, can also be merged for matching operations. The precondition of the merged operation in the composed model is the disjunction of the preconditions associated with the matching operations, and the postcondition of the merged operation is the conjunction of their postconditions.

The merging of the `writeLine()` operations in the primary and context-specific aspect models produces an operation that calls the buffer's write operation `writeBuff()` and the filestream's write operation `addToStream()`. This is not the desired result: The intent is to completely decouple `Writer` from `FileStream` using `WriteBuffer`. To resolve this problem, the following composition directives can be used:

- A pre-merge composition directive that removes the association between `Writer` and `FileStream` in the primary model.
- A pre-merge composition directive that removes the operation specification associated with the `writeLine()` operation in the primary model.

Once the above pre-merge directives are applied the composition algorithm is used to compose the modified primary model with the context-specific aspect model.

As another example, consider the partial context-specific and primary class models shown in Fig. 4. The `addUser()` operation in the primary model adds a user (instance of `User`) to a collection of users (instance of a class `User Repository`). The `addUser()` operation in the context specific aspect model calls the `doAddUser` operation only when the client calling the operation is authorized to add a user. The `doAddUser()` operation adds a user to the collection. Using signatures that consist only of model element names, the two *Repository Manager* classes match and thus their properties are merged. During the merge of these two classes, the `addUser()` operations are matched and their specifications (not shown) are merged. The resulting `addUser()` operation specification will have a semantic conflict: The specification from the

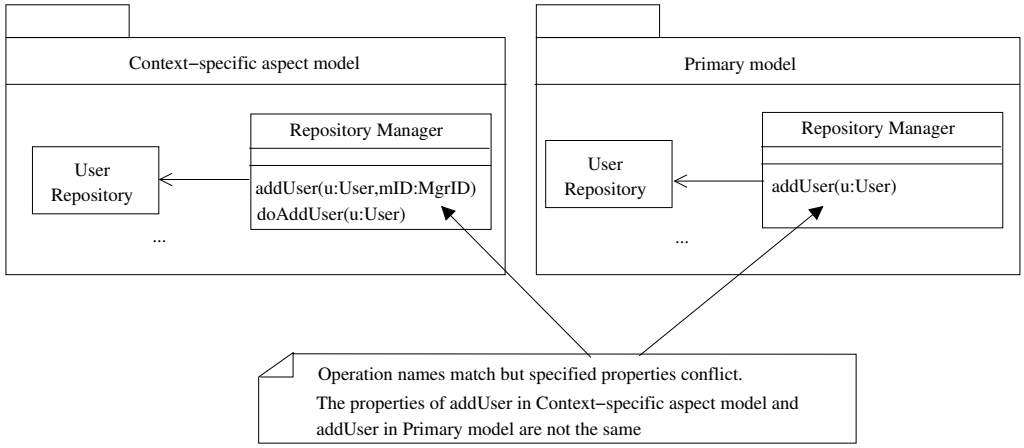

Fig. 4. Example of a Property Conflict

primary model allows unconditional adding of users, but the specification from the context-specific model will allow adding of users only if the operation is authorized for the client. This is an example of a *semantic property conflict*: A semantic property conflict occurs when two matching elements (elements with the same signature) are associated with conflicting semantic properties. In this example, the intent is to merge the `doAddUser()` operation in the context specific aspect model with the `addUser()` operation in the primary model. To resolve this conflict and reflect the intent, a pre-merge composition directive that renames the `addUser()` operation in the primary model to `doAddUser()` can be used. After this renaming, signature-based composition will produce a composed model with the required properties.

Renaming directives can also be used to resolve *syntactic naming conflicts*. A syntactic naming conflict occurs when two or more model elements representing different concepts have the same name. This class of conflicts can be avoided by instantiating the generic aspect model such that the names do not match or by using a pre-merge rename directive.

In some cases, post-merge directives are needed to add or delete elements in the model produced by merging primary and context-specific aspect model to produce a model that has required properties. For example, associations may be added between a class introduced by the primary model and another class introduced by a context-specific aspect model to provide required access to behaviors defined in the classes, or they may be removed to prevent access that is to be prohibited in the composed model.

With the ability to rename, add, and remove elements comes the risk of another type of conflict: The nonexistent-reference conflict. A nonexistent-reference conflict arises when a reference in one of the models refers to an element that no longer exists, or exists under a different name. To resolve this conflict, the affected references in a model must be identified and updated. Composition directives that identify and update specified references are needed.

In an aspect-oriented model that contains multiple aspect models, different composition orderings may produce different composed models [6]. A particular ordering can lead to undesirable emergent behaviors. For example, consider an auditing feature and a password feature that are to be composed with a primary model. If the password feature is composed with the primary model before the auditing feature, then the end result could be a model in which the auditing feature captures and stores passwords. This may be an undesirable emergent behavior. Composition directives that can be used to specify the order used to compose multiple aspects with a primary model are needed.

Defining composition ordering raises another type of conflict. A cyclic-ordering conflict occurs when there is a cycle among ordering relationships defined over multiple aspects. Analysis can detect and correct ordering conflicts.

The above discussion indicates that the following list of actions should be captured by composition directives:

- Creating new elements.
- Adding elements to a Namespace.
- Deleting elements from a Namespace.
- Changing property values of elements.
- Finding and changing references to specified model elements.
- Specifying override relationships between matching elements.
- Changing default composition rules
- Specifying ordering relationships among multiple aspects.

The above list of actions reflects our current experience and may be incomplete.

3 The Composition Metamodel

Our composition metamodel uses static and behavioral features needed to support model composition. In this paper, we describe the behavioral properties in terms of class operations and narrative descriptions of the operations. Alternatively, sequence and activity diagrams can be used to describe the interactions and activities that take place during composition.

The core part of the metamodel has been implemented using Kermeta, an open source meta-modeling language developed by the Triskell team at IRISA [7]. KerMeta extends the *Essential Meta-Object Facility* (EMOF) 2.0 [8] with an action language that allows one to describe the behavior of operations associated with classes in a metamodel. Kermeta was used primarily because it is compatible with the Eclipse Modeling Framework (EMF), which allows us to use Eclipse tools to edit, store, and visualize models manipulated in our AOM approach. A more detailed description of the language is presented in [9].

EMOF 2.0 is a subset of the Meta-Object Facility (MOF) that can be used to describe metamodels using object-oriented concepts. It utilizes concepts from UML 2.0, and thus allows one to use UML tools to build metamodels. EMOF defines a class called *Object* from which all other EMOF classes inherit properties. This class contains the following operations that will be used in the composition metamodel described later in this section:

- The $getMetaClass()$ operation returns the Class of an object.
- The $container()$ operation returns the containing parent object.
- The $equals(element)$ determines if the element (an instance of Element class) is equal to this Element instance.
- The $set(property, element)$ operation sets the value of the property to the element.
- The $get(property)$ operation returns a List or a single value depending on the multiplicity.

The $isComposite$ attribute defined in the EMOF class $Property$ returns true if the object is contained in the parent object. Cyclic containment is not possible, i.e. an object can be contained in only one other object. The $getAllProperties()$ operation in the EMOF class called $Class$ returns all the properties (including inherited properties) associated with a $Class$ object.

Fig. 5 shows the core part of the composition metamodel. The metamodel contains elements from the UML metamodel [2], but it differs from the UML metamodel in that it includes operations that specify composition behavior.

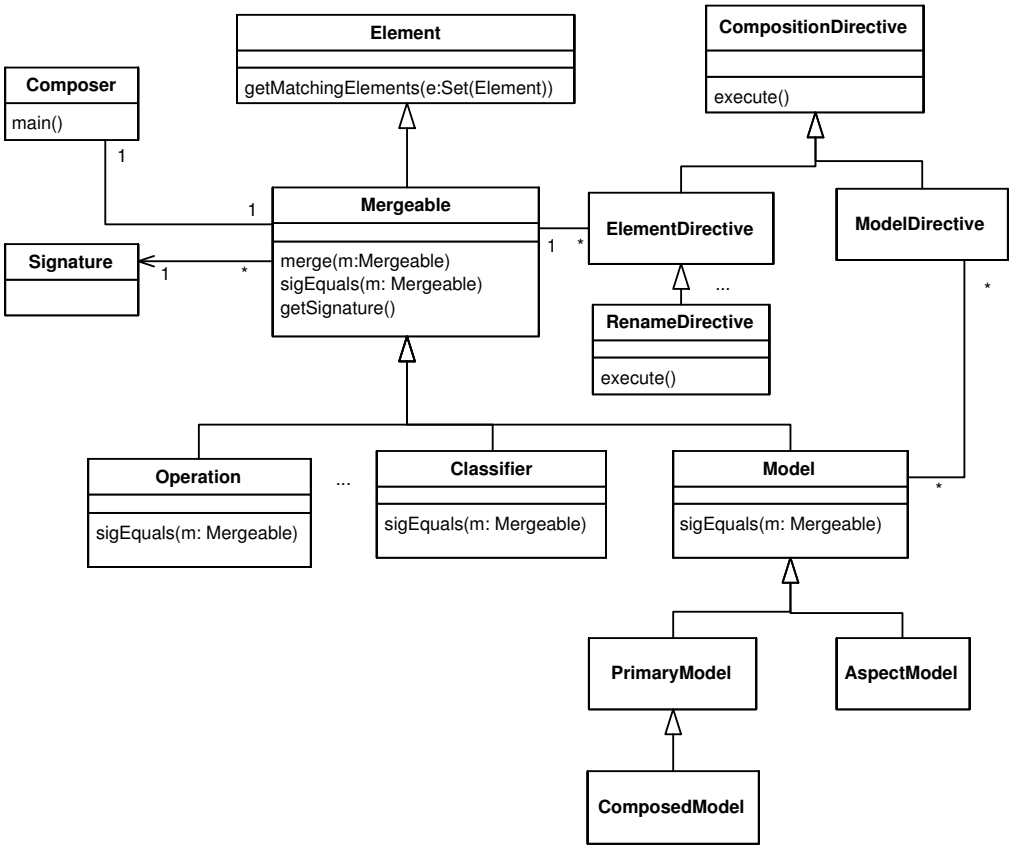

Fig. 5. Core Elements of Composition Metamodel

The core concepts shown in Figure 5 are described below:

– **Element**: Instances of this class are model elements. *Element* is an extension of the UML meta-class, $Element$. It is extended by the operation $getMatchingElements(e : Set(Element))$. Operations associated with the EMOF $Object$ class are also available in the *Element* class.
  - **Element::getMatchingElements()**: This operation takes in a set of elements and returns a set of elements that have the same syntactic type and signature as the element that invokes it. The syntactic type check is performed by invoking the $getMetaClass()$ and the $getAllProperties()$ operations defined in the EMOF $Object$ class. The signature is obtained using the $getSignature()$ operation.
– **Mergeable**: This is an abstract class that characterizes model elements that can be merged. Examples of mergeable elements shown in the figure are instances of $Classifiers$, $Operations$, and $Models$.
  - **Mergeable::merge()**: This operation merges the element with the mergeable element passed in as an operation argument. The merge method returns a new element that is the merge of the element m and the element on which the merge is called.
  - **Mergeable::sigEquals()**: This operation determines whether the element's signature is equal to the signature of another element.
  - **Mergeable::getSignature()**: This operation gets the signature of the element.
– **Signature**: Instances of this class are representations of signatures. Every mergeable element is associated with exactly one instance of this class.

The Kermeta implementation of the core parts of the composition metamodel (i.e., the metamodel obtained by excluding the *CompositionDirective* hierarchy) treats the model elements and instances of the other classes in the metamodel as objects (i.e., instances of the EMOF *Object* class). The implementation is thus written independently of model element types and it uses reflection to obtain type information. The operations in the composition metamodel (including those defined in EMOF) were implemented using the KerMeta action language.

The model elements are merged only when they have the same syntactic type and the same signature. The *sigEquals*() operation is used to determine whether signatures of model elements are the same (see appendix). Each model element type defines its own procedure for checking equality of signatures, that is, specializations of Mergeable can override the inherited *sigEquals*().

Merging of two matching model elements, $e1$ and $e2$, in the absence of composition directives proceeds as follows:

- **Primitive property rule:** A primitive property is a model element property that must be associated with exactly one value. The *isAbstract* property of classes is an example of a primitive property. The primitive properties of matching elements must have the same values. If they have different values then a conflict is indicated for each conflicting value. For example, if $e1$ and $e2$ are matching classes with different values for the *isAbstract* property then a conflict is indicated.
- **Composite property rule:** This rule applies to model element properties that are associated with values that are collections of model elements. The *ownedAttribute* property of a class is an example of this kind of property. This rule has a base case part and a recursive part. The recursive part essentially applies the merge recursively to merge the constituent parts of the property that match across the encompassing two model elements. The base case part determines the stopping condition for the recursion. In what follows, the composite property is referred to as p, $e1.p$ refers to the collection of values associated with p in $e1$ and $e2.p$ refers to the collection of values associated with p in $e2$.
  - **Recursive part:** For each constituent element in $e1.p$ a search is made for a matching element in $e2.p$ (based on the signature type associated with the constituent element type). If a match is not found then the element is included in merged form of $e1$ and $e2$. If a match is found the two matching constituent elements are merged and included in the merged form of $e1$ and $e2$.
  - **Base Case part:** If two constituent matching elements, $c1$ and $c2$, are composites that consist of only one model element, q, then the following occurs. If the signatures of $c1.q$ and $c2.q$ then $c1.q$ is merged with $c2.q$. If the signatures do not match then a conflict is indicated. For example, if two attributes are matched using only their names, then a conflict is indicated if their types do not match.

The composition of two models (instances of *Model*) is started by calling the *merge*() operation in one of the models, using the other as an argument. The *main*() method of the *Composer* class invokes the initial merge. Since a Model is not a primitive type, its *merge*() operation will result in the merging of the matching parts of the model. The algorithm for merging elements is given in the appendix.

Two types of composition directives are described in the composition metamodel. Element directives (instances of *ElementDirective*) are composition directives that apply to a group of elements in a single model. These directives can be used to add new elements, delete existing elements, rename elements, override elements, and replace references in a model. Model directives (instances of *ModelDirective*) are composition directives that are associated with a group of models. An example of a model directive is a composition directive that specifies the order in which aspects are composed with a primary model.

Each composition directive is associated with a behavior that implements the action associated with the directive. These behaviors are invoked by the *merge*() operations of elements before the merges of constituent properties are attempted.

The Kermeta implementation of the composition metamodel currently does not support the use of composition directives. We are now developing such support. The pre- and post-merge directives can be viewed as transformations on models and this is how they will be implemented in Kermeta (Kermeta was originally designed to support specification of model transformations).

4 Composition Directives

In this section we describe the composition directives that we have identified through application of the composition procedure on small case studies (e.g., see [10–12]). The directives can be used to modify aspect and primary models, add new elements to composed models or to override default composition rules in order to produce desired composed models. The directives that modify models can be viewed as transformations on the models. Directives that affect only aspect and primary models are applied to the models before their elements are merged. Those that add elements to composed models and those that override composition rules are applied during merging.

Each directive (except for the directives that override composition rules) is described using the following format:

- *Directive Name:* This section states the name of the directive or the form of names for a family of directives.
- *Application:* This section describes the purpose of the directives and describes the entities that the directives operate on.
- *Form:* This section describes the syntactic form of the directives.
- *Constraint:* This section gives the conditions that must hold if the directives are to have the intended effect. The constraint in this section is referred to as the directive precondition.
- *Effect:* This section describes the effect of the directives on their targets. The specification of effect is called the directive postcondition.

As indicated in the composition metamodel described in the previous section, there are two types of composition directives: Element directives and model directives. The following subsections describe the directives in each of these categories and gives examples of their application.

4.1 Element Directives

We have identified the following element directives thus far:

- Creating new model elements (a family of directives)
- Adding model elements to a namespace
- Removing model elements from a namespace
- Changing properties (a family of directives)
- Replacing references to a model element in a namespace
- Overriding model elements
- Overriding composition rules (a family of directives)

When an element is created by a create directive, a handle that can be used to reference the element is provided. These handles can be used in composition directives that are applied after the creation of the model elements. The names that appear on model elements in aspect and primary models serve as references to the model elements in directives. For example, an association name or a role name can refer refer to an association in a directive.

Creating new model elements. The following describes the family of create directives.

Directive Name: **create**<metamodel class name>
The following are examples of names for create directives: **create**Association, **create**Class, where *Association* and *Class* are the names of concrete classes in the UML metamodel.

Application: The create directives are used to create new model elements (i.e., model elements that are not in the primary or aspect models being composed). In the composition metamodel, each concrete *Element* class is associated with a constructor. The create directives use these constructors to create model elements to ensure that the created elements are syntactically well-formed. The new element is not a member of any namespace when it is created.

A create directive has set of operands that determines the arguments passed to the constructors of the model elements. The operands are a set of (*property name* = *property value*) pairs, where the *property name* is the name of a model element property.

Form: newHandle = **create**<Element> {operands}

The following is an example of a create directive that creates a concrete class with a name "NewClass".

```
newClass = createClass {name = "NewClass", isAbstract = false}
```

The following create directives are used to create a strong aggregation relation between two existing classes: primary::UserMgmt, and aspect::UserAuth.

```
userAuthEnd = createProperty { isComposite = false, aggregation = none,
  type = aspect::UserAuth, opposite = userMgmtEnd, lower = 1, upper = 1 }

userMgmtEnd = createProperty { isComposite = true, aggregation = composite,
  type = primary::UserMgmt, opposite = userAuthEnd, lower = 1, upper = -1 }

userAuth-userMgmt = createAssociation { name = "UserAuth-UserMgmt" ,
  isDerived = false, memberEnd = [userAuthEnd,userMgmtEnd] }
```

The operands of the above directives indicate that the two association ends (property) userAuthEnd and userMgmtEnd must be created before the association userAuth-userMgmt is created. We assign the value of "-1" to upper (representing the upper limit of a multiplicity) where "-1" represents the multiplicity "*". The "[..]" notation is used to denote a collection of association ends in the **create**Association directive.

Constraint: There are no constraints for these directives.

Effect: A create directive provides a reference to a new model element that is valid. The new Element is not a member of any namespace.

Adding model elements to a namespace.

Directive Name: **add**

Application: The **add** directive is used to add a model element to a namespace in a model. It can be used to add a newly created model element (i.e., one created by a create directive) to a namespace and to add an element from another namespace into a target namespace. The latter action is needed when a model element is migrated to a new namespace in order to ensure that the composed model has required properties. Such a migration would involve removing the element from its original namespace (using the **remove** directive described later) and then adding it to the new namespace.

The **add** directive has one operand, the model element to be added.

Form: **add** `owner::elem`
In the above, the model element, `elem` is added to the namespace, `owner`.

Constraint: The target namespace must exist, the element to be added must have a unique name within the namespace, and the element must be an instance of a concrete UML metamodel class that can be owned by the namespace.

Effect: The element is in the target namespace.

Removing model elements from a namespace.

Directive Name: **remove**

Application: The **remove** directive is used to remove a model element from a namespace. It is used when the presence of certain model elements compromises desired properties of the composed model. For example, consider a security aspect model that requires that certain associations not exist in the composed model because their presence can lead to leaks of sensitive information. The **remove** directive can remove these associations in the primary model.

Removing a composite model element involves removing all its contained parts. For example, removing an association involves removing its association end properties (but not the classes at the association ends).

Removing a model element can result in models with hanging references: References to the removed element may be present in the namespace and elsewhere (e.g., in OCL expressions) after removal. Use of the directive should be coupled with the use of other directives that take care of the hanging references. For example, one can use the **replaceOccurrences** directive to replace reference to the deleted element with references to other elements.

The **remove** directive has one operand, the model element that is to be removed.

Form: **remove** `owner::elem`
In the above, the model element, `elem` is removed from the namespace, `owner`.

Constraint: The namespace must exist in a model. The element must be in the namespace before the directive is applied.

Effect: The element is not in the namespace.

Changing properties of model elements in a namespace. The family of directives for changing model element properties are described below.

Directive Name: **change**<property name>
Examples of change directive names are **change**isAbstract, and **change**name. The **change**name directive is written more concisely as **rename**.

Application: The **changeProperty** directive is used to change the value of a model element property. This directive can be used to force or prevent matching of model elements by changing the property values used to determine element matches. For example, in the cases where matching is based only on the names of elements, this directive can be used to rename elements so that they match or do not match.

This directive has two operands. The first is the model element with the property, the second is the new value of the property.

In our case studies we often use this directive to rename model elements and thus we use a more concise name for the directive: **rename**. The renaming directive is often applied to the primary model, because renaming of elements in the context-specific aspect models can also be accomplished by rebinding the (generic) aspect model.

Form: **change**<property name> owner::targetElement **to** propertyValue

In the cases where the property to be changed is a model element name one can use the form below:

rename owner::targetElement **to** newName

Constraint: The element must exist in a primary, aspect or composed model.

Effect: The specified property value in the target model element has the new value.

Replace references to a model element in a namespace.

Directive Name: **replaceOccurrences**

Application: The **replaceOccurrences** directive is used to replace references to a model element with references to another model element in a namespace. It is often used in conjunction with directives that add and remove model elements. For example if an association that is referenced in an OCL expression is removed then one can use this directive to change the reference in the OCL expression.

The **replaceOccurrences** directive has two operands: The first is a reference to a model element, and the second is a reference to another model element.

Form: **replaceOccurrences** owner1::elem **with** owner2::replacementElem
The above states that references to elem in the namespace owner1 are to be replaced by references to replacementElem in the namespace owner2.

Constraint: There are no constraints for this directive.

Effect: All existing references to the model element owner1::elem are changed to references to the element owner2::replacementElem.

Overriding a model element. This composition directive is similar to the override relationship proposed by Clarke et al. [13].

Directive Name: **override**

Application: The **override** directive defines an override relationship between two potentially conflicting model elements. It indicates that the properties of a model element takes precedence over properties of a matching model element during composition.

When an **override** relationship is defined for two model elements, the relationship propagates to the contained model elements. The consequences of the implicit overrides may not be immediately obvious. Explicit **override** relationships should be defined for contained model elements when this is feasible and practical.

The **override** directive has two operands. The second operand is the model element that overrides the first operand.

Form: **override** `owner1::elem1` **with** `owner2::elem2`

Constraint: `owner1::elem1` and `owner2::elem2` must exist in separate models, one in a primary model, and the other in a context-specific aspect model. The two elements must match.

Effect: During composition, the properties of `elem1` are replaced by properties of `elem2`.

Overriding default composition rules. When merging matching model elements with different property values, a composition mechanism can use default rules to determine the property values that will be used in the composed model. For example, in previous work [5] we defined the following rules for combining properties with different values in matching elements:

- If two matching attributes are associated with invariants, the invariant in the composed model is the conjunction of the two invariants.
- If two matching operations have operation specifications, the composed operation has a precondition that is the disjunction of the two preconditions and a postcondition that is the conjunction of the two postconditions.
- If two associations match and their multiplicities are different, then the merged association uses the weaker multiplicity constraint at each end.

Sometimes one may want to change the default rules when composing models. For example, one may want to use the stronger multiplicity constraint at the ends of composed associations. *Override composition rule* directives are used for this purpose. In our approach, each rule is associated with a set of possible variations and a directive for each variation is defined. For example, the association end multiplicity rule is associated with the following directive:

association end multiplicity rule `owner1::assocend1; owner2::assocend2` **stronger**

Use of this directive indicates that the stronger of the two multiplicities at the specified associations are to be used in the composed model. One can also override the rule globally using the following directive:

association end multiplicity rule stronger

For the operation specification rule we have the following directive:

operation specification rule `owner1::aclass1::`**PreSpec**`(anoperation1),`
`owner2::aclass2::`**PreSpec**`(anoperation2)` **conjunct**

The above states that the precondition of the operation formed by merging the matching operations `anoperation1` and `anoperation2` is the conjunction of their preconditions. A similar directive for postconditions is also defined:

operation specification rule `owner1::aclass1::`**PostSpec**`(anoperation1),`
`owner2::aclass2::`**PostSpec**`(anoperation2)` **disjunct**

Currently we have a very limited number of composition rules. In the cases where we do not have such rules, composition results in a conflict when the property values differ. Work on providing a small and useful set of rules and associated directives is ongoing.

4.2 Composition Examples

The following are examples of composition scenarios that require the use of directives to produce desired results. In the examples we show the effect of directives in terms of before and after diagrams. Note that the after diagrams are not the composed models: They show only the effect of the directives on the primary and aspect models.

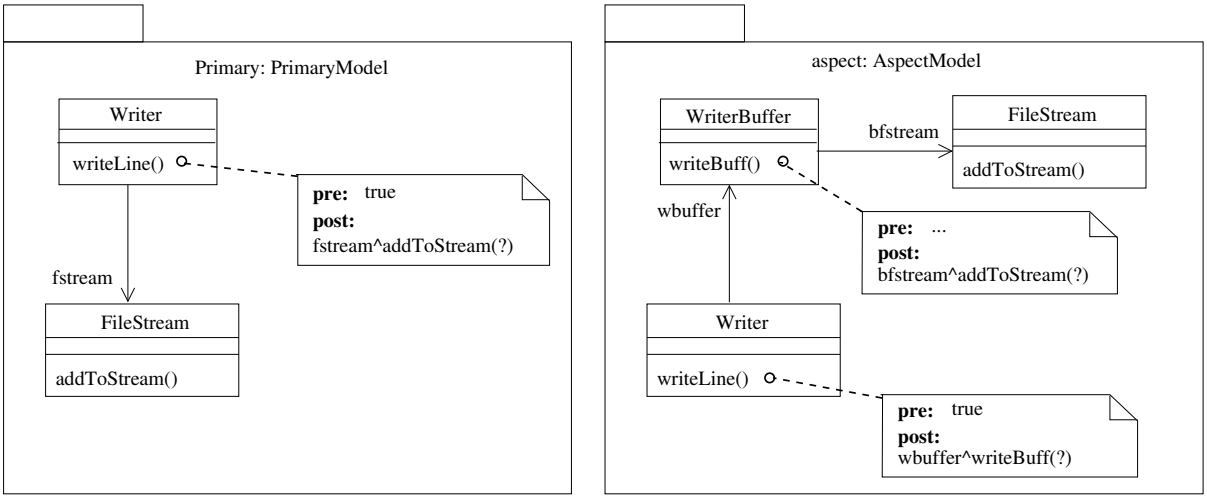

Fig. 6. Example 1. Before Application of directives.

Example 1: The faulty composition shown in Fig. 3 can be avoided by using composition directives that do the following (the aspect and primary models are shown in Figure 6):

1. Remove the association between `Writer` and `FileStream` in the primary model: In the desired composed model, all writing to the file stream is done via the buffer. The write should not have direct access to the filestream in the composed model.

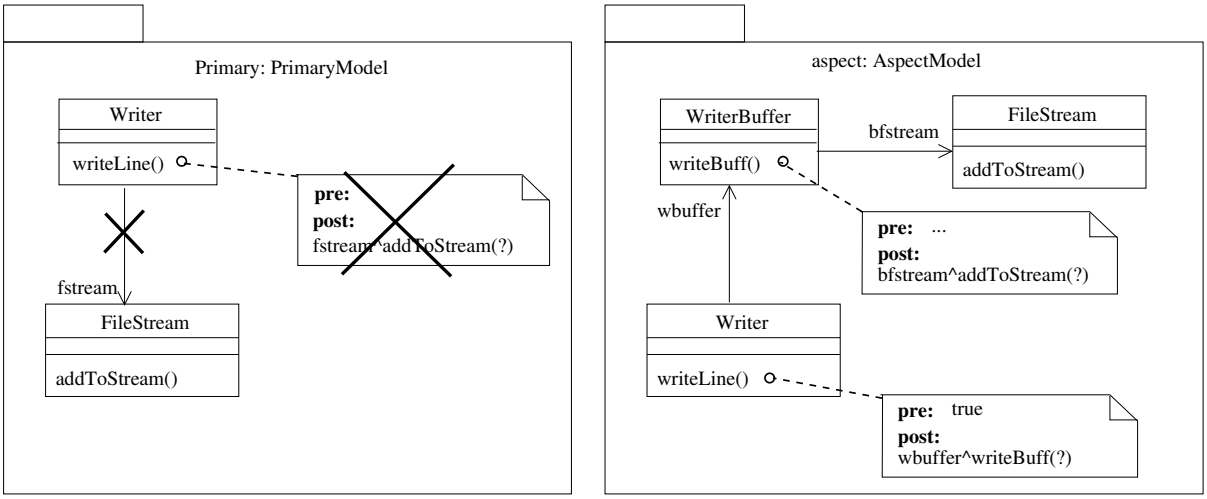

Fig. 7. Example 1. After Application of remove directives.

2. Remove the OCL specification for `writeLine()` in the primary model: The operation specification in the context-specific aspect model fully specifies the desired behavior and thus the conflicting specification in the primary model can be deleted.

   The directives that accomplish the above are given below:

   (1) **remove** `primary::Writer::fstream`
   (2) **remove** `primary::Writer::`**Spec**`(writeLine)`

In the above, **Spec**(writeLine) refers to the specification associated with the operation `writeLine()`. Figure 6 and Figure 7 illustrate the effect of the directives on the primary and aspect model. An "X" indicates the removal of an element.

In the example, the operation specification associated with `writeLine()` in the primary model contained only a statement that refers to the deleted *fstream* element. If the specification had contained additional statements that were required in the operation specification of `writeLine()` in the composed model, then removal of the specification in the primary model would not give the desired result. To handle these situations, directives that replace the elements to be removed in the OCL specifications with desired elements are needed. Such directives require technology for parsing OCL expressions. A metamodel for the OCL is currently being standardized by the Object Management Group (see http://www.omg.org/uml) and it is expected that OCL parsers based on the metamodel will be developed soon after.

An alternative way to accomplish the above would be to use the **override** directive instead of the second **remove** directive as shown below.

(1) **remove** `primary::Writer::fstream`
(2) **override** `primary::Writer` **with** `aspect::Writer`

Figure 8 illustrates the effect of the directives on the primary and aspect models.

Example 2: The following example, from France et al. [1], illustrates the use of the **create**, **add**, **remove** and **replaceOccurrences** directives. The aspect model shown in Figure 9 presents a view in which add and delete

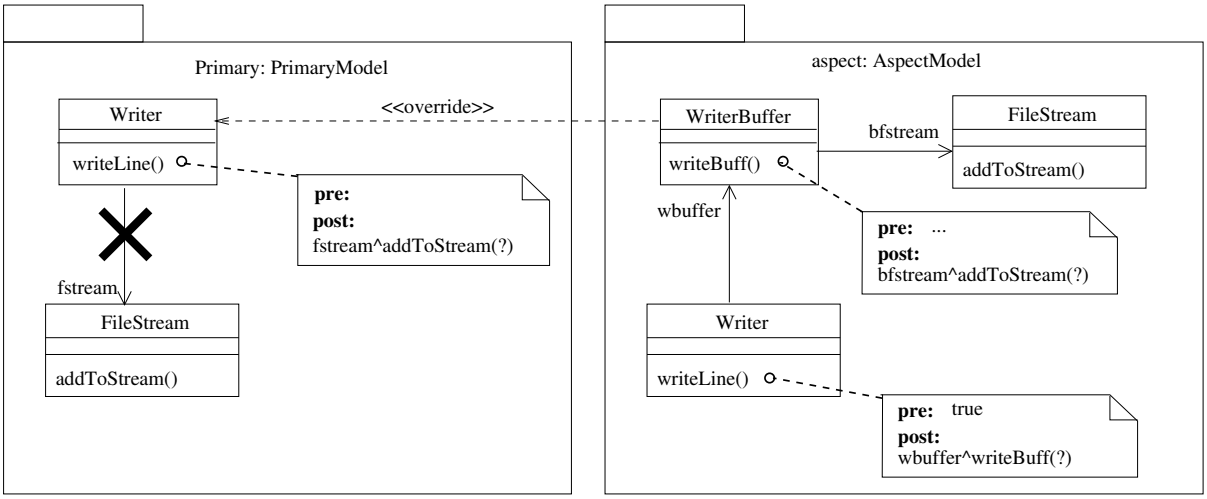

Fig. 8. Example 1. After Application of remove and override directives.

user actions must be authorized before they are carried out. The primary model describes a view in which authorization does not occur. The objective of the composition is to produce a composed model in which the authorization behavior in the aspect is incorporated into the primary model. In Figure 9, the `UserAuth` class in the aspect model performs authorization checks on clients requesting the addition or deletion of users from the system. In the composed model, `Manager` client must request the add and deleter user operations by calling the corresponding operations in `UserAuth` and should have no direct access to the `UserMgmt` class. To accomplish this, a directive is used to remove the `accesses` association in the primary model:

(1) **remove** `primary::Manager::accesses`

There are references to the `accesses` association in `Manager` that must be replaced or removed. In this case, references to `accesses` in the primary model must be changed to `uaccesses` in the context specific aspect model, because all access to the operations is made via the `uaccesses` association in the composed model. The following directive is used to accomplish this:

(2) **replaceOccurrences** `primary::Manager::accesses` **with** `aspect::Manager::uaccesses`

The definitions of the `addUser` and `deleteUser` operations in `UserAuth` include an authorization check. In the aspect model, if a `Manager` client is authorized to carry out the add or delete action a call is made to the respective `doAddUser`, `doDeleteUser` operations. In the described composed model, the operations `addUser` and `deleteUser` in `UserMgmt` carry out the add and delete user actions, respectively. To make this possible a composition directive that adds an association between the `UserMgmt` class and the `UserAuth` class is used:

(3) `userAuthEnd = createProperty { isComposite = false, aggregation = none, type = aspect::UserAuth, opposite = userMgmtEnd, lower = 1, upper = 1 }`

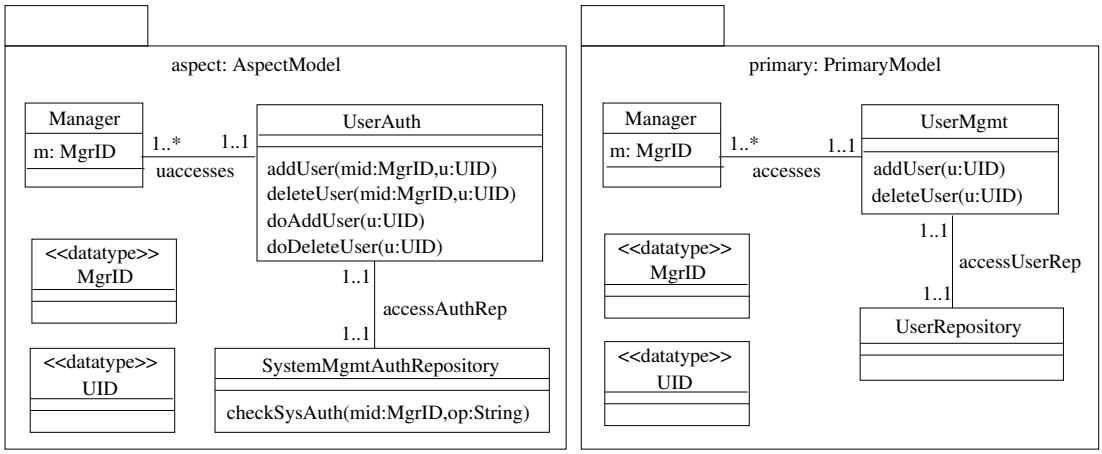

Fig. 9. Example 2. Before application of directives.

```
userMgmtEnd = createProperty { isComposite = true, aggregation = composite,
  type = primary::UserMgmt, opposite = userAuthEnd, lower = 1, upper = -1 }

userAuth-userMgmt = createAssociation { name = "UserAuth-UserMgmt" ,
  isDerived = false, memberEnd = [userAuthEnd,userMgmtEnd] }
```

Once the new `Association` is created, we need to add it to the composed model. The composition directive that accomplishes this is given below. We reference the composed model using the name *comp*:

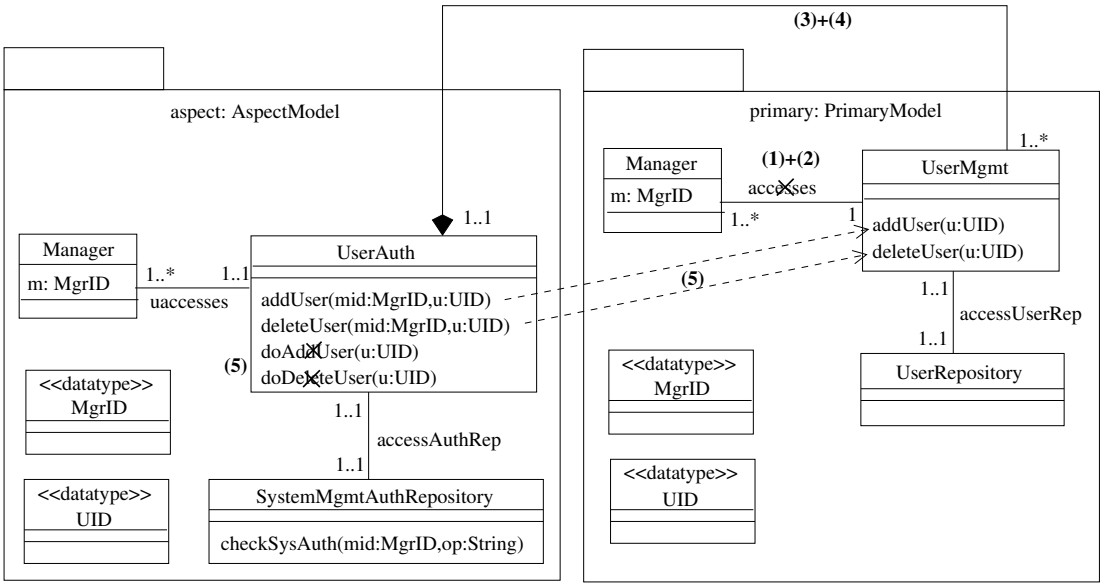

Fig. 10. Example 2. After application of directives.

(4) **add** `comp::userAuth-userMgmt`,
add `comp::UserAuth::userAuthEnd`,
add `comp::UserMgmt::userMgmtEnd`

There are two options for creating a composed model in which authorized calls to `addUser` and `DeleteUser` are made: The first option is to replace the specifications of `doAddUser` and `doDeleteUser` so that they delegate the actions to the respective operations in `UserMgmt` using the new association. The second option is to replace the calls to `doAddUser` and `doDeleteUser` by calls to the respective operations in `UserMgmt`. We give the directives that accomplish the latter option below:

(5) **replaceOccurrences** `aspect::UserAuth::doAddUser`
with `primary::UserMgmt::addUser()`,
remove `aspect::UserAuth::doAddUser`,
replaceOccurrences `aspect::UserAuth::doDeleteUser`
with `primary::UserMgmt::deleteUser()`,
remove `aspect::UserAuth::doDeleteUser`

The effect of the directives on the aspect and primary models is shown in Figure 10. The association between `UserMgmt` and `UserAuth` exists in the composed and not in the aspect or primary models - it is shown here only to indicate that this association will exist in the composed model. The dependencies from the `addUser` and `deleteUser` operations in `UserAuth` indicate that they call the respective operations in `UserMgmt`.

4.3 Combining Element Directives

The examples and the descriptions of composition directives provide some indication that use of some element directives are often coupled with the use of others. For example, removing a model element sometimes requires use of directives such as the **replaceOccurrences** directive to avoid hanging references. An overview of combined directives in the pre-merge, merge and post-merge categories are given below:

Pre-Merge Combined Directives: Matching directives are combined directives that force the matching of elements or disallow the matching of elements. The directives are often combinations of **changeproperty** and **replaceOccurrences** directives.

Merge Combined Directives: Combinations of the **override** and **replaceOccurrences** directives are often used to override rules used to merge model elements.

Post-Merge Combined Directives: These directives are often combinations of directives for creating model elements, adding model elements to a namespace and deleting model elements from a namespace.

The development of a library of combined directives that are based on actual use of directives on realistic projects is a major goal of our research on composition directives.

4.4 Model Directives

Model directives determine how a set of models are composed. The model directives we have identified constrain the order in which context-specific aspect models are composed with a primary model. These directives can define a weave-ordering relationships between aspect models. A weave-ordering relationship is a binary constraint that specifies an ordering between two aspect models. There are two cases: An aspect model must be composed before another, or an aspect model must be composed after another.

Precedes

Directive Name: **precedes**

Application: This directive specifies that one aspect model is to be composed with a primary model before another. This directive has two aspect models as operands. The first operand is the aspect model that is to be composed the second operand.

Form: `former` **precedes** `latter`

Constraint: Both aspect models must exist.

Effect: A weave-ordering relationship is created between the two aspect models, and added to the set of weave-ordering constraints maintained by the composer. This directive does not imply that `former` will be woven immediately before `latter`. It simply requires that `former` be woven some time before `latter`.

Follows

Directive Name: **follows**

Application: This directive specifies that one aspect model is to be composed with a primary model after another. This directive is provided only to increase the readability of composition directives. It may be interpreted as equivalent to the **precedes** directive with the operands switched. This directive has two aspect model operands. The first operand is the aspect model to be composed after the second operand.

Form: `later` **follows** `earlier`

Constraint: See **precedes**.

Effect: See **precedes**.

4.5 Weave Ordering Example

Consider the aspect design model in Figure 11(a). There are three different aspect models and the primary model. In this example, the authentication aspect model needs to be composed before the authorization aspect model, because authorization without authentication is meaningless. Therefore, we declare the following composition directive to make the order explicit.

(1) `authentication` **precedes** `authorization`

We could have also defined a composition directive using the **follows** directive with the operands reversed to achieve the same result.

Suppose we also wish to weave the `errorChecking` aspect model last. The following composition directives accomplish this:

(2) `errorChecking` **follows** `authorization`
(3) `errorChecking` **follows** `authentication`

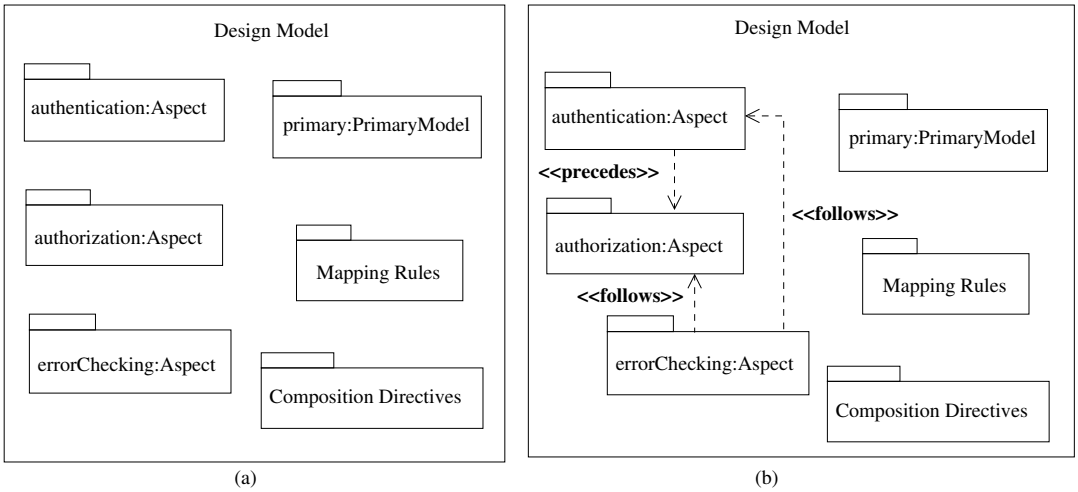

Fig. 11. Example 4. Specifying Weave Order

The result is shown in Figure 11(b). The dependency from authentication to authorization illustrates the weave-order relationship that specifies that authentication must be woven before authorization, and the dependencies from errorChecking to each of the other aspects illustrates the two binary weave-order relationships that specify errorChecking as the last aspect to be woven.

5 Related Work

A number of researchers have developed aspect oriented software development (AOSD) approaches (e.g., see [13–20]). The composition procedures used in these AOSD approaches can be categorized as *asymmetric* and *symmetric* [21]. In *asymmetric* composition, aspects and base models play clearly distinguished roles during composition. These composition approaches tend not to support composition of aspects and composition of base models. AspectJ [22] is one of the popular aspect-oriented programming languages that uses an asymmetric composition procedure. In *symmetric* composition both aspect and base models are treated the same and thus aspect and base model composition are possible. The composition approaches used in work on viewpoints [23], subject-oriented programming [24, 25], and multi-dimensional separation of concerns (MDSOC) [26] tend to be symmetric. The composition approach outlined in this paper uses a hybrid composition procedure: The (generic) aspect models are patterns that cannot be directly composed with base models, but the instantiated forms of the aspect models (i.e., context-specific aspect models) are not distinguished from the primary model by the composition procedure. The model composition procedure we developed can be used to compose (generic) aspect models (i.e., patterns) to obtain new aspect models (e.g., see [27]) and to compose UML models. To date we have implemented the procedure for composing UML class models.

A survey of AOSD approaches can be found in Chitchyan et al. [28]. Very few approaches in the survey provide support for composing design models. At the programming level, the subject-oriented approach is closest to the approach described in this paper. In subject-oriented programming [24, 25], program elements such as classes and methods are composed by merging corresponding elements. The correspondence is established based on specified composition rules. The default correspondence is name-based, which can be altered by writing additional composition rules. The composition rules used to control this process can be classified under three categories: rules that establish correspondence, rules that control combination, rules

that control both correspondence and combination. The composition rules in subject-oriented programming are analogous to our use of signatures to determine matches and the use of directives to alter model elements and override default composition rules. Our composition procedure depends on the properties specified in the signature rather than just names of model elements, primarily because not all UML model elements are named elements. We have found that name-based matching has a greater potential of producing faulty models than signature-based composition, simply because signature-based composition allows for finer tuning of matching criteria.

At the model level, a comparable AOM approach is the Theme approach proposed by Baniassad and Clarke [13, 29, 30]. In the Theme approach, a design, called a *theme*, is created for each system requirement. These themes, like context-specific aspect and primary models, are essentially design views. A comprehensive design is obtained by composing themes. Composition in the Theme approach is based on the symmetric approach used in subject oriented programming. Composition relationships specify how models are to be composed by identifying overlapping concepts and specifying how models are integrated. Two types of integration strategies are used: Override and merge. Override integration is used when existing behavior in a subject needs to be updated to reflect new requirements. Merge integration is used when subjects for different requirements are to be integrated. Operations in related subjects may need to be merged into a unified operation. Reconciliation strategies resolve conflicts between property values of corresponding subject elements. Precedence relationships, transformation functions applied to conflicting elements, explicit specification of reconciled elements, and default values may be used for reconciliation. Clarke [13] also extends the UML metamodel with the notion of *composableElements* that can be composed using a composition relationship. They have a *Match* metaclass that supports specification of matching criteria. Their matching criteria includes *matchByName* and *dontMatch*. They leave the details of implementing the *matchByName* and *dontMatch* to the user of the metamodel. In this sense the metamodel describes a framework for composing UML models. In our work we have developed a more specialized metamodel that contains specifications of composition behaviors. The metamodel was designed to describe our composition procedure and to guide the development of supporting tools. To validate the metamodel, we used it to develop a prototype tool for composing UML class models. The composition directives that we have developed include some that are similar to the merge and override integration strategies. The use of composition directives and signatures, as described in this paper, allow modelers to define and apply their own integration and reconciliation strategies, and thus gain finer control over how models are composed.

Brito and Moreira describe an aspect composition process that identifies match points in a design element and defines composition rules [31]. Rules use identified match points, a binary contribution value (either positive or negative) that quantifies the affects on other aspects, and a priority for a given aspect. In the context of AOP [32], Kienzle et al. describe composition rules based on dependencies between aspects [33]. Both papers [31, 33] focus primarily on relationships that can exist between aspects. We describe the possible relationships between aspects as weave-order relationships and override relationships, but it may also be possible to use priorities and dependencies as done by Kienzle, Brito and Moreira in our approach. In this sense, the ideas presented in their papers complement the ideas presented in this paper.

Aldawud et al. [34] propose a mechanism for composing state charts where a crosscutting behavior as an event that triggers a state transition. The composition is specified by linking events across state diagrams. We have not considered composition of state charts in our work.

6 Conclusions and Future Work

In this paper we present a signature-based composition approach that allows one to vary how models are composed using composition directives. The signature-based approach improves upon name-based compo-

sition approaches by giving the modeler finer-grained control over the criteria used to match model elements. Composition directives give added flexibility by providing the means to alter model elements and override default composition rules to obtain desired composed models. The directives described in this paper are based on our experience with using the composition approach to compose aspects modeling security features with primary models. For example, we have applied the approach to modeling and composition of access control features such as Role-Based Access Control and BLP schemes [5, 27, 35, 36], and for other security features [6, 37–39]. We are currently applying the techniques in a larger case study involving the development of an E-Commerce system.

A composition metamodel that describes the static and behavioral properties needed to support model composition is also presented. The metamodel describes not only the static relationships among composition concepts, but also provides specifications of behaviors that are needed to support model composition using our approach. The composition metamodel describes the behavior needed to support model composition and thus can be used to guide the development of model composition tools that support the composition approach we developed. To validate the metamodel, we built a prototype tool on top of the KerMeta framework. The tool currently supports the composition of UML class models and can be extended to support additional features that appear in the composition metamodel. We are currently developing a subsystem for handling composition directives that will be plugged into the tool.

Empirical evaluation is needed to validate the composition approach in real world design settings. Such studies can determine the amount of effort required to specify the kinds of compositions that are required in real world designs. The studies can also be used to determine whether the composition directives match the requirements of a real project. The insights gained from the studies will be used to develop a tractable method for selecting, defining, and applying composition directives and signatures. Work in this respect could result in the specification of some common composition strategies [6] to ease the task of specifying and using composition directives.

Acknowledgement

This material is based upon work partially funded by AFOSR under Award No. FA9550-04-1-0102.

References

1. France, R.B., Ray, I., Georg, G., Ghosh, S.: An aspect-oriented approach to design modeling. IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design **151** (2004) 173–185
2. The Object Management Group (OMG): Unified Modeling Language: Superstructure. Version 2.0, Final Adopted Specification, OMG, http://www.omg.org (2003)
3. Straw, G., Georg, G., Song, E., Ghosh, S., France, R., Bieman, J.: Model composition directives. In: Proceedings of the International Conference on the UML, October 2004, Springer (2004) 84–97
4. Reddy, R., France, R.B., Ghosh, S., Fleury, F., Baudry, B.: Model composition - a signature based approach. In: Proceedings Aspect Oriented Modeling workshop held with MODELS/UML 2005, Montego Bay, Jamaica (2005)
5. Song, E., Reddy, R., France, R., Ray, I., Georg, G., Alexander, R.: Verifiable composition of access control and application features. In: SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies, New York, NY, USA, ACM Press (2005) 120–129
6. Georg, G., Ray, I., France, R.: Using Aspects to Design a Secure System. In: Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002), Greenbelt, MD, ACM Press (2002) 117–126
7. TRISKELL: The KerMeta Project Home Page. URL `http://www.kermeta.org` (2005)
8. OMG Adopted Specification ptc/03-10-04: The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, (http://www.omg.org)
9. Muller, P., Fleury, F., Jézéquel, J.: Weaving executability into object-oriented meta-languages. In: Proceedings of MODELS/UML 2005, Montego Bay, Jamaica (2005)

10. Reddy, Y.R., France, R.B., Georg, G.: An aspect-based approach to modeling and analyzing dependability features. Technical Report CS04 - 109, Colorado State University (2004)
11. France, R., Georg, G.: Modeling fault tolerant concerns using aspects. Technical Report 02-102, Computer Science Department, Colorado State University (2002)
12. Georg, G., France, R.B., Ray, I.: Composing aspect models. In: 4th AOSD Modeling with UML workshop, San Francisco, CA (2003)
13. Clarke, S.: "Extending Standard UML with Model Composition Semantics". Science of Computer Programming **44** (2002) 71–100
14. Araujo, J., Coutinho, P.: Identifying aspectual use cases using a viewpoint-oriented requirements method. In: Early Aspects 2003: Aspect Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd Intl. Conference on Aspect-Oriented Software Development, Boston, MA (2003)
15. Clarke, S., Walker, R.J.: Composition Patterns: An approach to Desigining Reusable Aspects. In: Proc. of 23rd Intl. Conference on Software Engineering (ICSE), Toronto, Canada (2001) 5–14
16. Gray, J., Bapty, T., Neema, S., Tuck, J.: Handling crosscutting constraints in domain-specific modeling. Communications of the ACM **44** (2001) 87–93
17. Grundy, J.C.: Multi-perspective specification, design and implementation of software components using aspects. International Journal of Software Engineering and Knowledge Engineering **20** (2000)
18. Jacobson, I.: Case for Aspects - Part I. Software Development Magazine (2003) 32–37
19. Rashid, A., Sawyer, P., Moreira, A., Araujo, J.: Early aspects: A model for aspect-oriented requirements engineering. In: IEEE Joint Intl. Conference on Requirements Engineering, Essen, Germany (2002) 199–202
20. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. In Guerraoui, R., Nierstrasz, O., Riveill, M., eds.: Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming. Volume 791., Springer-Verlag (1994) 152–184
21. Harrison, W., Ossher, H., Tarr, P.: Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM - RC22685 (W0212-147) (2002)
22. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingier, J., Irwin, J.: Aspect oriented programming. In: Proc. of the European Conference on Object-Oriented Programming (ECOOP), Springer Verlag LNCS 1241, Finland (1997) 220–242
23. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. IEEE Transactions on Software Engineering **20** (1994) 760–773
24. Harrison, W., Ossher, H.: Subject oriented programming (a critique of pure objects). In: Proc. of the 8th Annual Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA '93), Washington, D.C. (1993) 411–428
25. Ossher, H., Kaplan, M., Katz, A., Harrison, W., Kruskal, V.: Specifying subject-oriented composition. Theory and Practice of Object Systems, Wiley and Sons **2** (1996)
26. Tarr, P., Ossher, H., Harrison, W., Sutton, S.: N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering (ICSE '99). (1999) 107–119
27. Ray, I., Li, N., Kim, D.K., France, R.: Using parameterized UML to specify and compose access control models. In: Proceedings of Sixth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2003). (2003)
28. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of aspect-oriented analysis and design approaches. Technical Report ULANC-9, AOSD - Europe (2005)
29. Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: Proceedings of the International Conference on Software Engineering. (2004) 158–167
30. Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: The 23rd International Conference on Software Engineering (ICSE), Toronto, Canada. (2001)
31. Brito, I., Moreira, A.: Towards a composition process for aspect-oriented requirements. In: Proceedings of the Early-Aspects Workshop at AOSD2002. (2002)
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97). Volume 1241 of Lecture Notes in Computer Science., Jyvaskyla, Finland (1997) 220–242
33. Kienzle, J., Yu, Y., Xiong, J.: On composition and reuse of aspects. In: Proceedings of the Foundations of Aspect-Oriented Languages Workshop, Boston, MA, USA (March 2003)
34. Aldawud, O., Bader, A., Elrad, T.: Weaving with statecharts. In: Workshop on Aspect-Oriented Modeling (held with AOSD-2002), Enschede, Netherlands (2002)
35. Ray, I., France, R., Li, N., Georg, G.: An aspect-based approach to modeling access control concerns. Information and Software Technology **40** (2004) 557–633

36. Ray, I., Li, N., France, R., Kim, D.K.: Using UML to visualize role-based access control constraints. In: Proceedings of the Symposium on Access Control Models and Technologies (SACMAT). (2004) 31–40
37. Georg, G., France, R., Ray, I.: Designing High Integrity Systems using Aspects. In: Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002), Bonn, Germany (2002)
38. Georg, G., France, R., Ray, I.: An Aspect-Based Approach to Modeling Security Concerns. In: Proceedings of the Workshop on Critical Systems Development with UML, Dresden, Germany (2002)
39. Homb, S.H., Georg, G., France, R., Bieman, J., Jurjens, J.: Cost-benefit trade-off analysis using bbn for aspect-oriented risk-driven development. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). (2005)

A Merge part of the signature-based composition procedure

```
*****************************************************************************
// e1 and e2 are the model elements that need to be merged
e1.merge(e2 : ModelElement)     //precondition : e1.sigEquals(e2) returns true
*****************************************************************************
result := e1.getMetaClass.new // create the merged instance in the context of e1

// Iterate on all properties of the objects to be merged.
// e1 and e2 have the same meta-class. Thus, they have the
// same set of properties.

foreach Property p in e1.getMetaClass.getAllProperties

  if type of p is primitive
  // Primitive types are basic datatypes such as string, int.
  // If an object does not have a value for a property then
        // the value val is taken from the other object and vice versa.
        // This is not a conflict.
  // If neither object has values, then val is null in the resulting
  // merged object.
     if e1.get(p) is null or e2.get(p) is null then
         result.set(p, val)
     else
         // If the values are the same then it is ok.
         // Otherwise a conflict has been detected.
         if e1.get(p) = e2.get(p) then
              result.set(p, e1.get(p))
         else
              A conflict has been detected
  else
  // Type of p is not primitive.
  // If the property refers to a single object, this is the base case.
     if the property upper bound is 1
         if e1.get(p) is null or e2.get(p) is null then
             result.set(p, val)      // val is the same as above
         else
             if sigEquals(e1.get(p), e2.get(p)) then
             // If the object e1.get(p) is contained by e1 and same for e2
             // (p.isComposite=true) then the objects should be merged,
             // otherwise, one is chosen.
             // Either one can be chosen because they both have the same signature
                   if p.isComposite is true then
                         result.set(p, merge(e1.get(p), e2.get(p)))
                   else
                         result.set(p, e1.get(p).clone())
```

```
            else
                  A conflict has been detected
      else
      // The property refers to a collection of objects.
      // The resulting merged object should contain property values that are
         // either only in e1 or only in e2, or the merged version of objects
         // that are in both e1 and e2.
         for each value v1 in e1.get(p)
             for each matching element v2 in e2.get(p)
                 if p.isComposite then
                     result.get(p).add(merge(v1, v2))
                  else
                     result.get(p).add(v1.clone())
                     if no element found
                        result.get(p).add(v1.clone())
             for each value v2 in e2.get(p)
                 if NO matching element found in e1.get(p)
                     result.get(p).add(v2.clone())
*******************************************************************************
```